Introduction to Information & Communication Technologies

CL-1000

Lab 07

Introduction to JavaScript

National University of Computer & Emerging Sciences – NUCES – Karachi

National University of Computer & Emerging Sciences – NUCES – Karachi

Course Code: CL-1000	Introduction to Information & Communication Technologies

Contents

1. Introduction

1.1 What is JavaScript

JavaScript is one of the most popular and widely used programming languages in the world right now. It's **growing faster** than any **other programming language**, and big companies like **Netflix**, **Walmart**, and **PayPal** build internal applications around JavaScript.

You can work as a **front-end developer**, a **back-end developer**, or even a **full-stack developer** who knows both the front end and the back end.

1.2 What can you do with JavaScript?

For a long time, JavaScript was only used in **browsers** to build interactive **web pages**. Some developers referred to JavaScript as a toy language, but those days are gone. Thanks to huge community support and investments by large companies like **Facebook** and **Google**, you can now build full-blown **web** or **mobile apps**, **real-time networking applications** like **chats** and **video streaming services**, **command-line tools**, and even **games**.

1.3 Where does JavaScript code run?

JavaScript was originally designed to run only in **browsers**, so every browser has a JavaScript **Engine** that can execute JavaScript code. For example, the JavaScript engines in **Firefox** and **Chrome** are **SpiderMonkey** and **V8**, respectively.

In 2009, a clever engineer named **Ryan Dahl** took the **Open-Source JavaScript Engine** in **Chrome** and embedded it inside a **C++ program**, which he called **Node.js**. Node.js is a C++ program that includes **Google's V8 JavaScript Engine**. With **Node.js**, we can now run **JavaScript** code outside of a browser, meaning we can use JavaScript to build the backend for web and mobile applications.

In a nutshell, JavaScript code can run inside a browser or in Node.js. Both browsers and Node.js provide a runtime environment for JavaScript code.

1.4 What is the difference between JavaScript and ECMAScript?

ECMAScript is just a **specification**, while JavaScript is a **programming language** that conforms to this specification. There is an organization called **ECMA** which is responsible for defining standards, including the **ECMAScript** specification.

The **first version** of **ECMAScript** was released in **1997**. Starting in 2015, ECMA has been working on annual releases of the newest specifications. **In 2015, they released ECMAScript 2015, also called ECMAScript version 6 or ES6 for short. This specification introduced many new features for JavaScript.**

2. Installing Node.js

Download Node.js LTS from the following link and install it. The installation in very simple, you have to just follow the installation steps.

Link: Node.js

3. Running JavaScript in Visual Studio Code

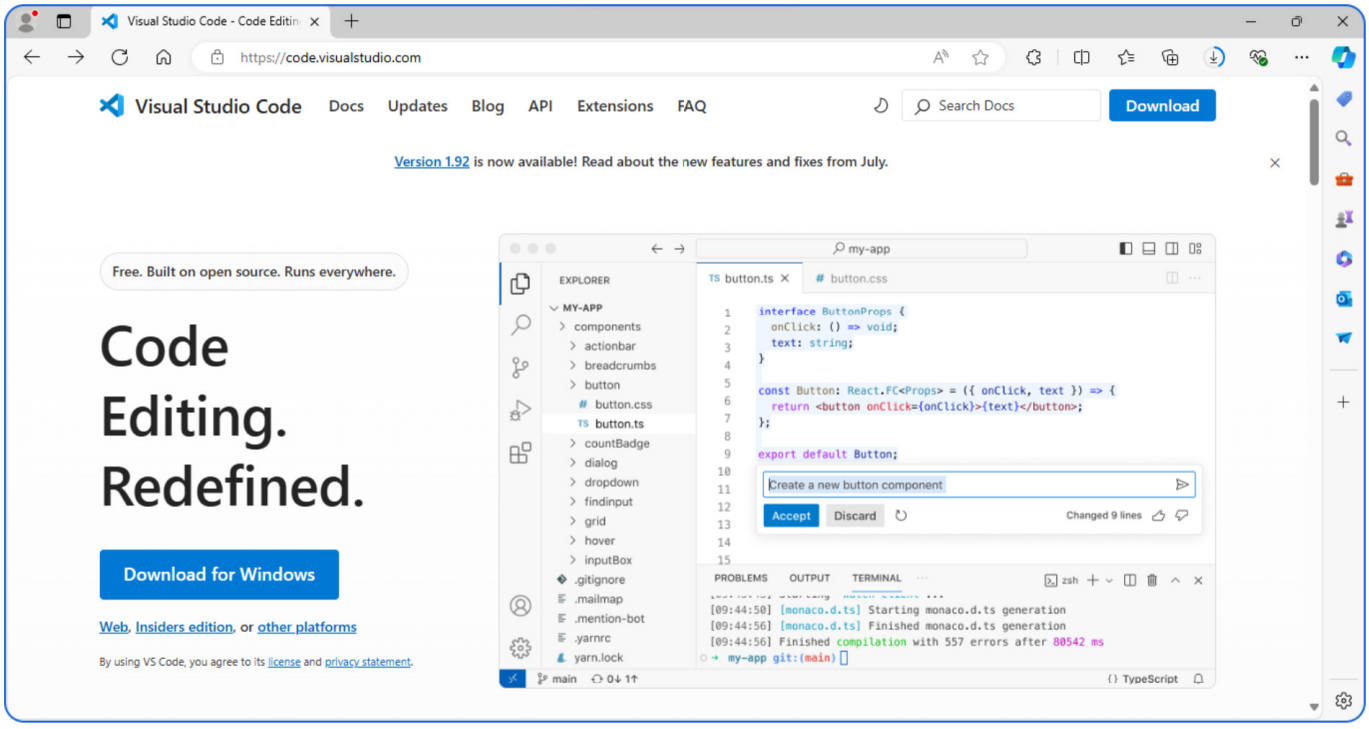
Visual Studio Code, also commonly referred to as **VS Code**, is a source-code editor developed by **Microsoft** for **Windows, Linux, macOS** and **web browsers**. Features include support for debugging, syntax highlighting, intelligent code completion, snippets, code refactoring, and embedded version control with Git. Users can change the theme, keyboard shortcuts, preferences, and install extensions that add functionality.

We will use VS Code for running our Python programs throughout this lab.

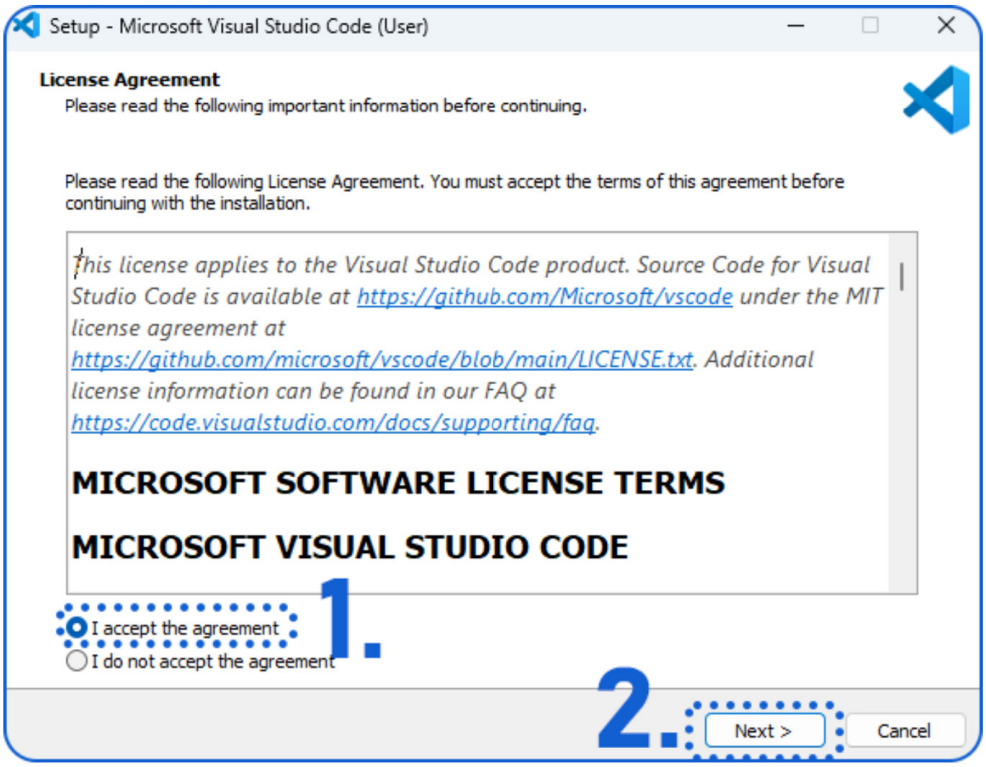
3.1 Installing Visual Studio Code on Windows

1. Click on the following link to download the Visual Studio Code installer.

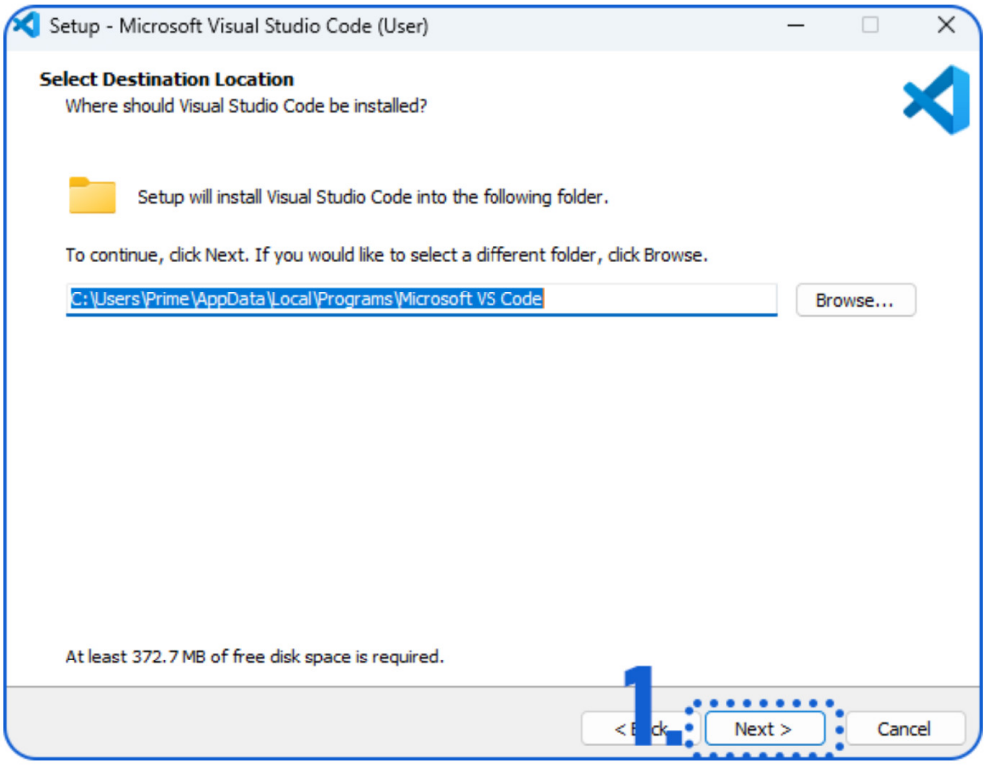
Link: **Visual Studio Code**



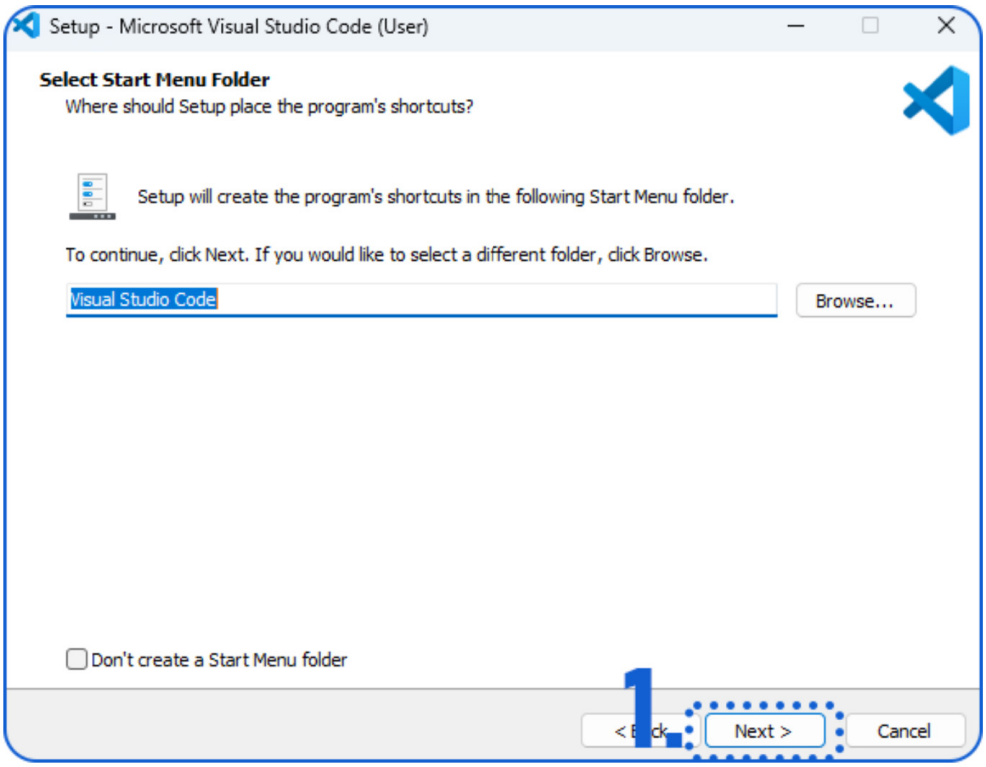
2. Run the downloaded installer.
3. Accept the License Agreement.
4. Click on **Next >**.



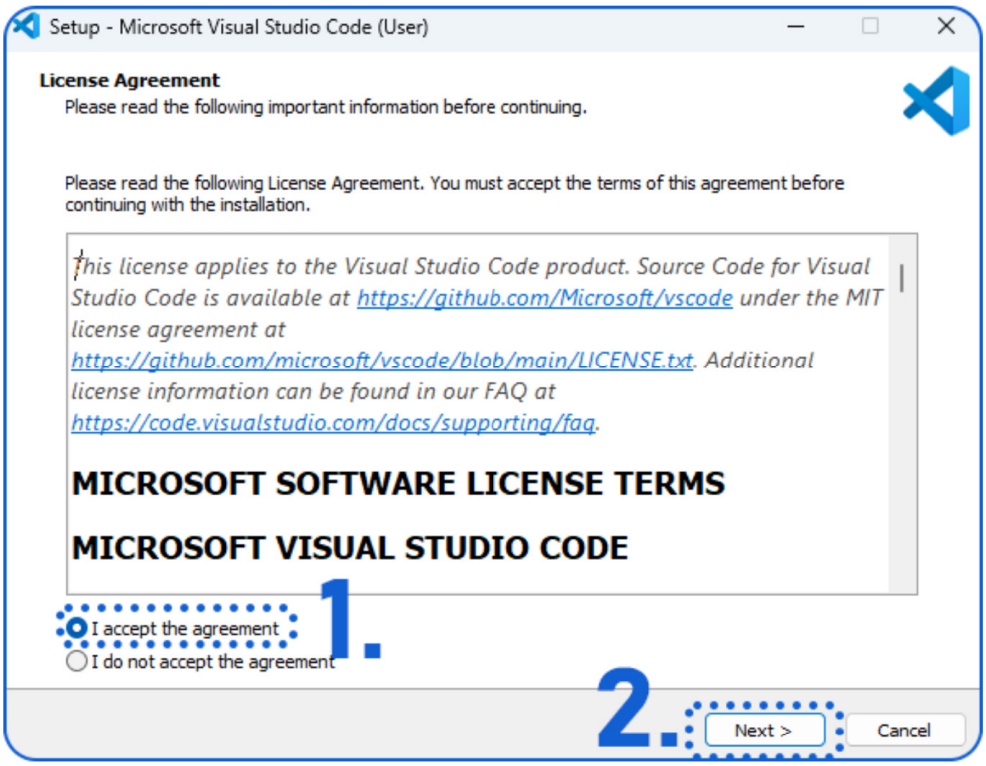
5. Leave the installation directory to default and click on **Next >**.



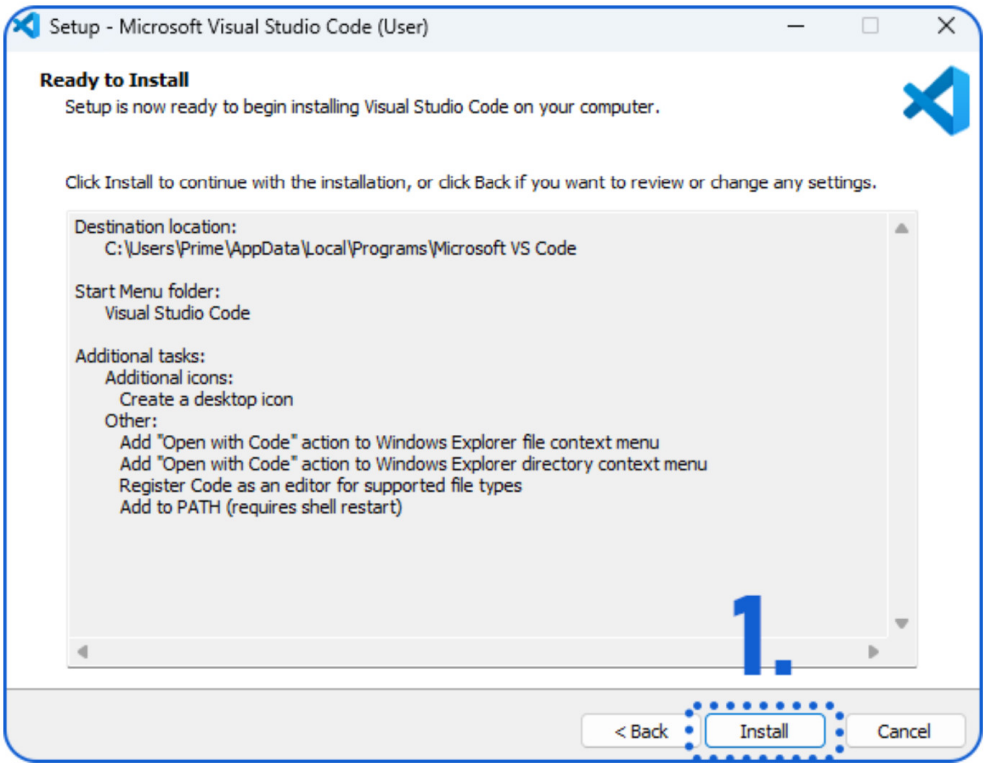
6. Click on **Next >**.

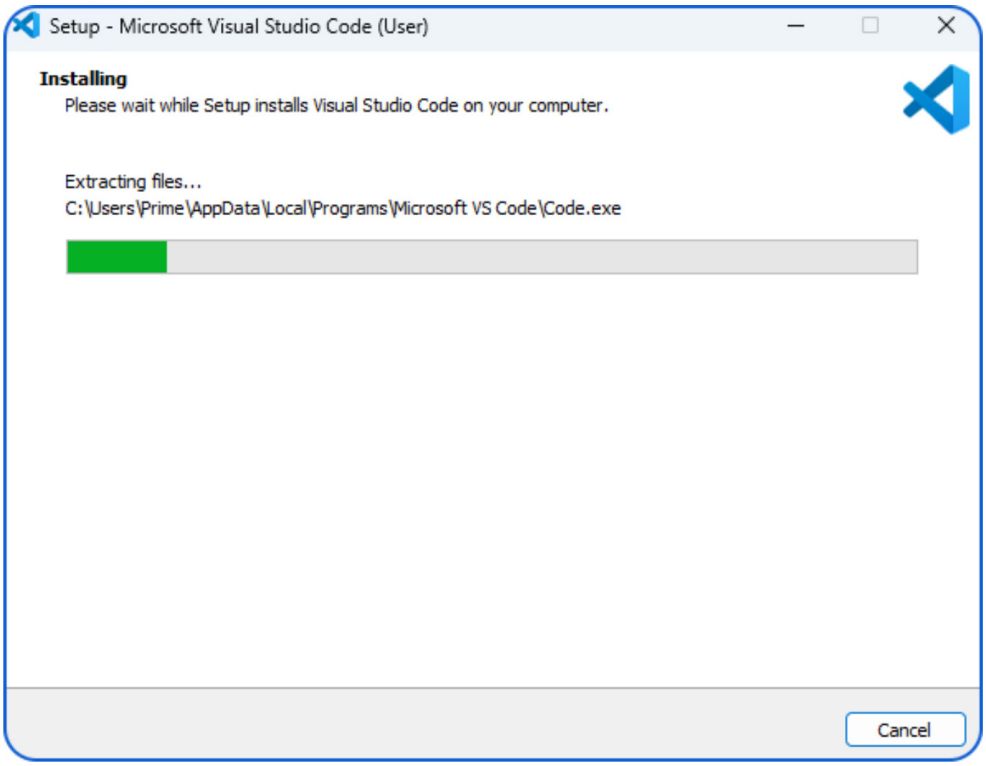


7. Check all the options and click on **Next >**.



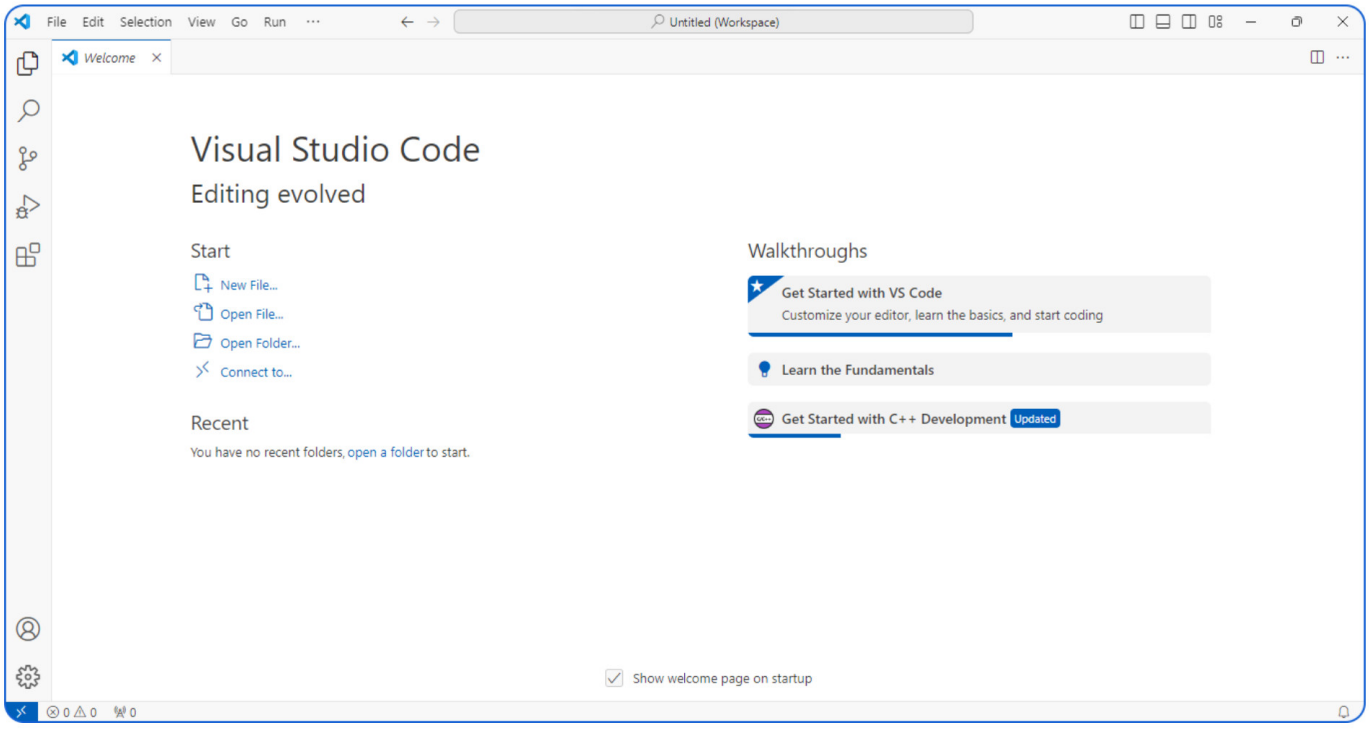
8. Click on **Install.**



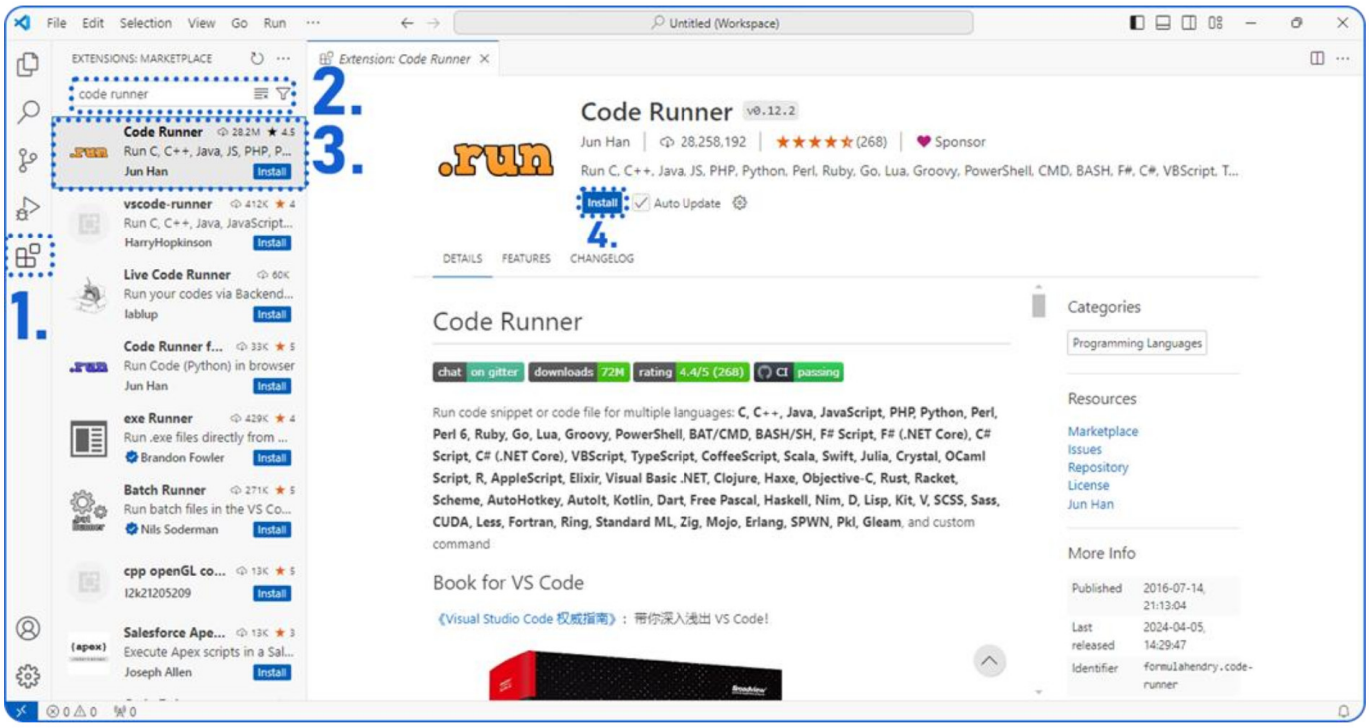


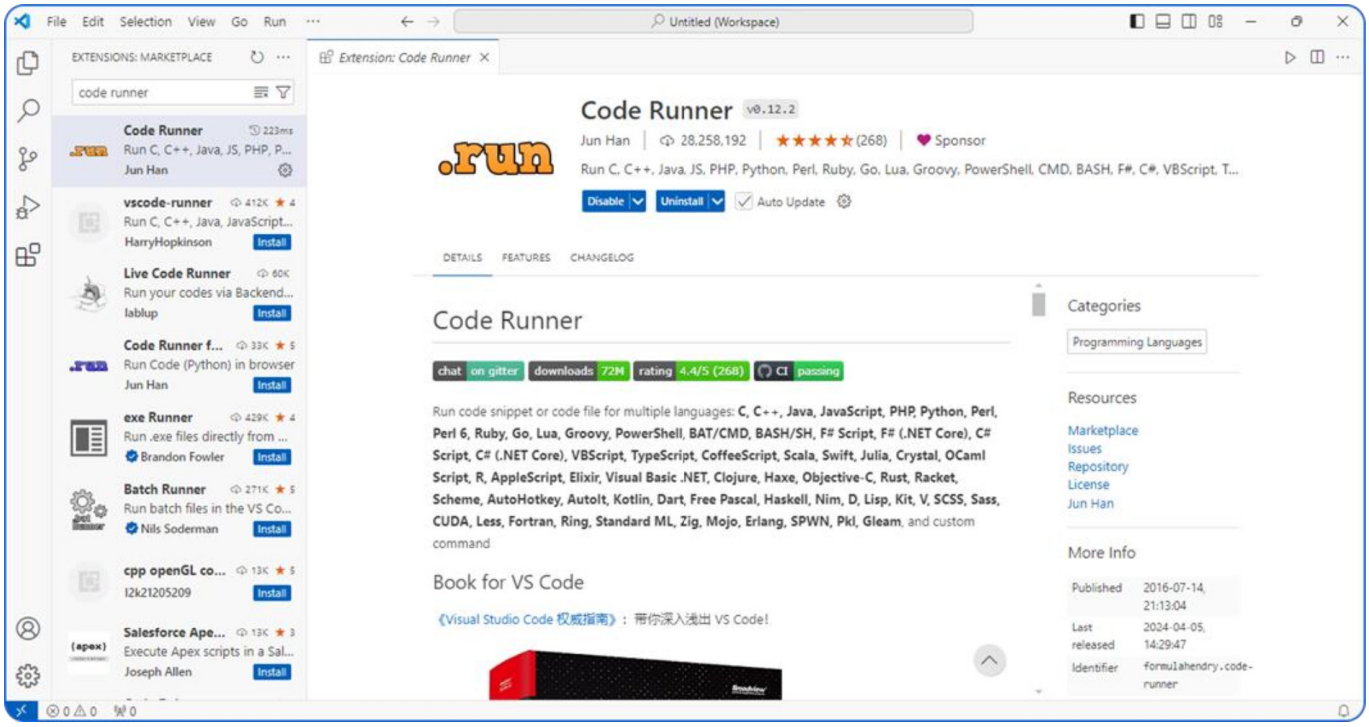
3.2 Configuring Visual Studio Code for JavaScript

1. Open Visual Studio Code.



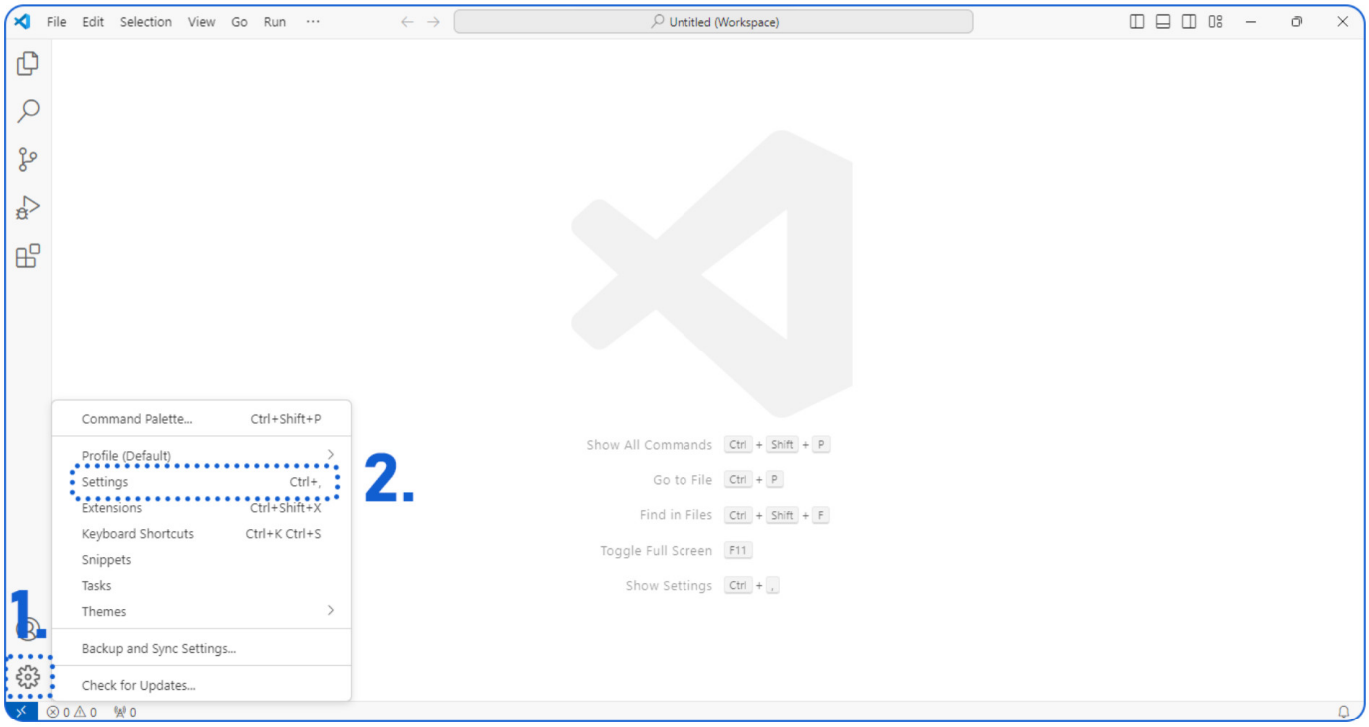
2. Click on the **Extension: Marketplace** icon at the left-most pane.
3. Type **code runner** and click on the **code runner** extension to install it.
4. The extension tab will open, click on the **install** button. Wait for the installation to finish.



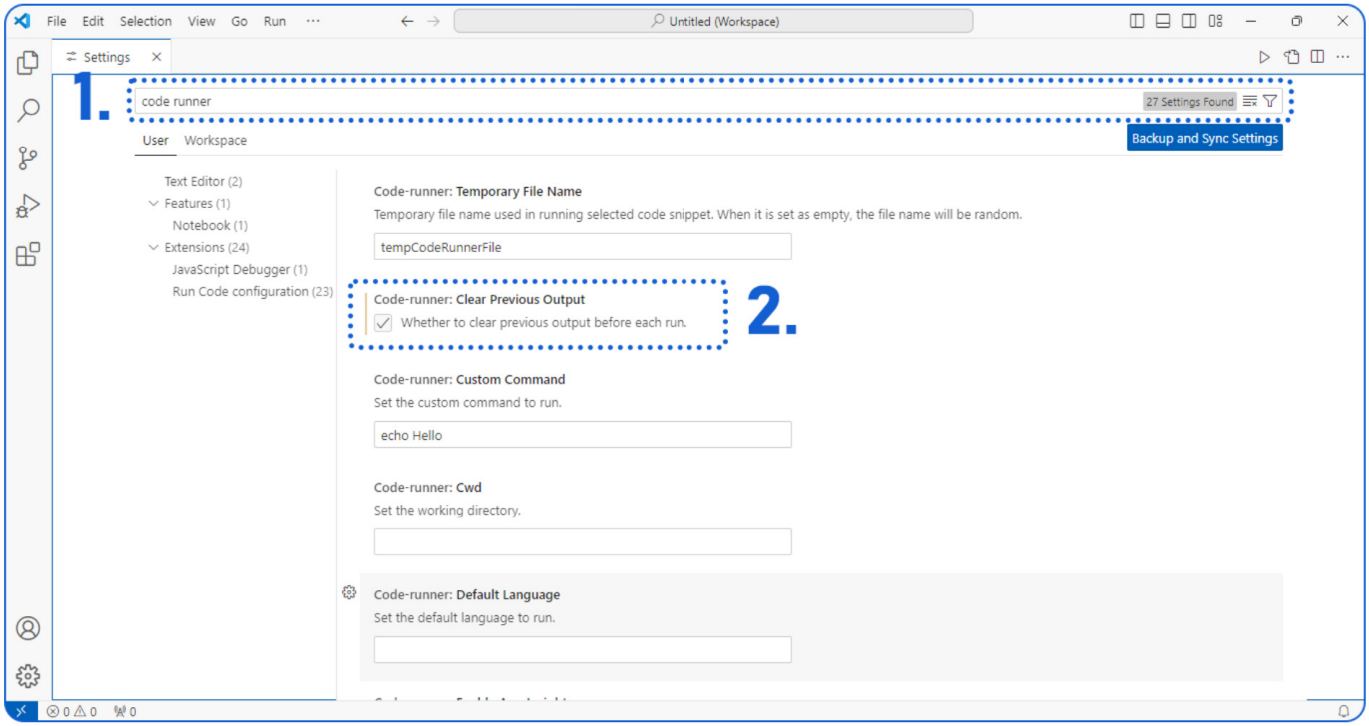


3.2.1 Setting -up Code Runner

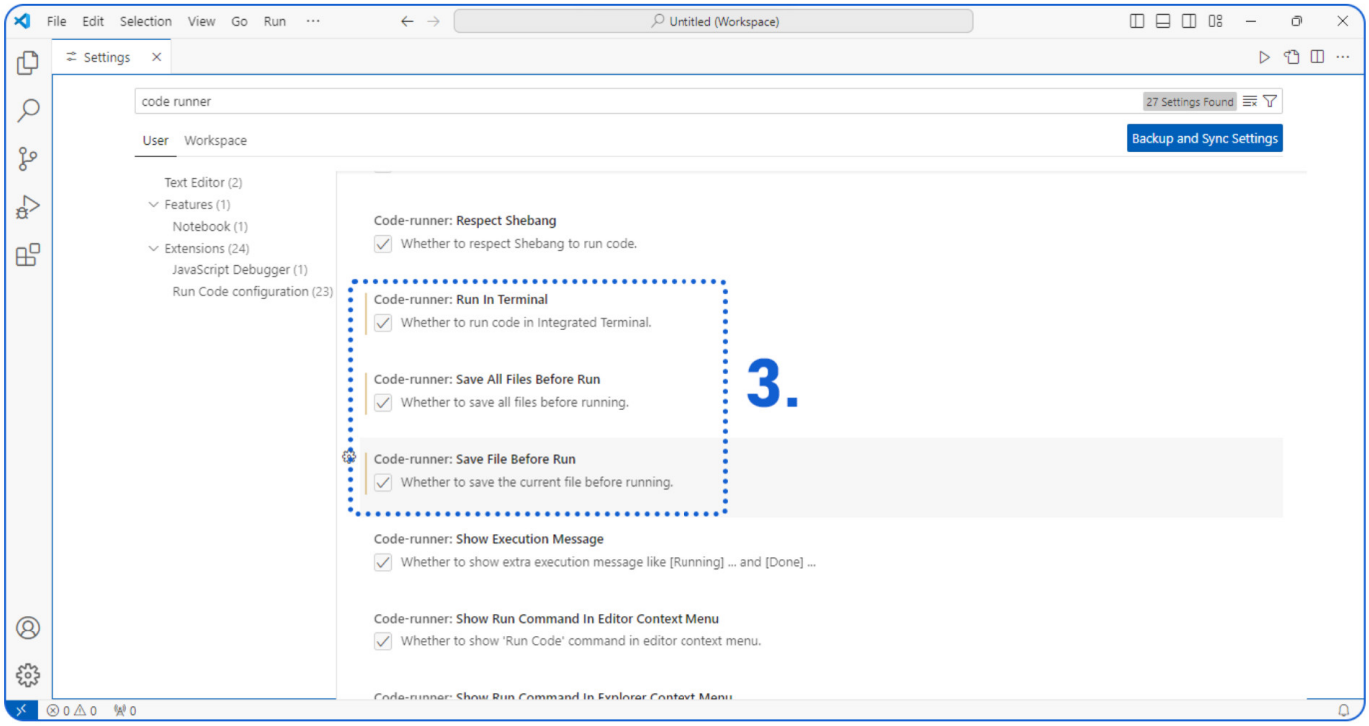
5. Close the extension tab after successful installation.
6. Click on the **setting** icon (the gear icon) at the bottom-left corner of the left pane.
7. Click on **Settings**. Or you can press **Ctrl +** ,.



8. Type **code runner** in the search bar and check **Whether to clear previous output before each run.**



9. Scroll down and check the following options.
    a. **Whether to run code in Integrated Terminal.**
    b. **Whether to save all file before running.**
    c. **Whether to save the current file before running.**



4. JavaScript in an HTML Document

Create a **`.html`** file in VS Code. Now, type exclamation mark in the file, then press **`Tab`** to generate basic HTML boilerplate. We'll use this as a host for our JavaScript code.

Open **`index.html`** with Live Server to launch it in your default browser. To ensure everything works, add an **`<h1>`** element in the body section with **`"Hello World"`** as the text. The page should refresh, showing **`"Hello World."`**

To add JavaScript, place a **`<script>`** tag at the end of the body section (after all HTML elements). This is a best practice because it allows the browser to fully render the page before running any JavaScript, preventing delays in loading content. Sometimes, third-party code requires placement in the head section, but generally, JavaScript should go at the end of the body.

You can now write your first JavaScript statement, `console.log("Hello World");`. This logs a message to the console. JavaScript statements should end with semicolons, and comments can be added using **`//`**. Comments explain why code is written a certain way rather than what it does, as that should be clear from the code itself.

To see the output of the JavaScript code, go to the developer tools in your browser and click on console.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0"
/>
    <title>Document</title>
  </head>
  <body>
    <h1>Hello World</h1>
    <script>
      // My first JavaSript Program
      console.log("Hello World");
    </script>
  </body>
</html>
```

4.1 Separation of Concern | Creating Separate File for JavaScript Code

Instead of writing JavaScript inline, best practices encourage separating HTML (content) from JavaScript (behavior). To do this, create a new file, index.js, move your JavaScript code there, and reference it in the HTML file using the **`src`** attribute in the **`script`** tag.

By following these steps, you've separated concerns, making your code more organized and maintainable.

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0"
/>
    <title>Document</title>
  </head>
  <body>
    <h1>Hello World</h1>
    <script src="index.js"></script>
  </body>
</html>
```

5. Variables and Constants

In JavaScript, variables and constants are used to store data values. JavaScript provides different ways to declare variables and constants, and understanding how to use them is crucial for writing efficient and effective code. Let's explore them in detail.

5.1. var (Variable Declaration - Before ES6)

Before ES6 (ECMAScript 6), the only way to declare variables in JavaScript was using **`var`.** A variable declared with **`var`** can be re-assigned and even re-declared within its scope.

Key Characteristics:

- **`var`** is function-scoped or global-scoped, depending on where it's declared.
- Variables declared with **`var`** are hoisted (i.e., they are moved to the top of their scope before code execution).

Example:

```javascript
var name = "John";  // Declare and assign a value
console.log(name);

name = "Doe";       // Re-assign a new value
console.log(name);

var name = "Alice"; // Re-declare and assign
console.log(name);
```

Output	John Doe Alice

Example: Hoisting

```
console.log(age); // Output: undefined (due to hoisting)
var age = 25;     // Declaration and assignment
```

Output	undefined

```
age = 25;           // assignment
console.log(age);
var age;            // Declaration
```

Output	25

5.2. `let` (Block-Scoped Variable - Introduced in ES6)

With ES6, **`let`** was introduced to declare variables that are block-scoped. This means the variable is only accessible within the block (e.g., a **`function`**, **`loop`**, or **`if`** statement) where it is defined. Unlike **`var`**, **`let`** does not allow re-declaration within the same scope but allows reassignment.

Key Characteristics:

- Block-scoped: Variables defined with **`let`** are scoped to the nearest enclosing block.
- Not hoisted like **`var`** (though technically hoisted, they are placed in the "temporal dead zone").
- Variables can be reassigned, but not re-declared within the same scope.

Example:

```
let score = 100;    // Declare and assign a value
console.log(score); // Output: 100

score = 200;        // Re-assign a new value
console.log(score); // Output: 200

// let score = 300; // SyntaxError: Identifier 'score' has already been
declared in this block
```

Output	100 200

Block Scope Example:

```
let greeting = "Hello";

if (true) {
    let greeting = "Hi";   // Block-scoped
    console.log(greeting); // Output: Hi
}

console.log(greeting);     // Output: Hello (outer scope)
```

Output	Hi Hello

5.3. `const` (Constant Declaration - Introduced in ES6)

`const` is used to declare constants in JavaScript. Once a constant is declared, its value cannot be changed or reassigned. Just like **`let`**, **`const`** is block-scoped. It is primarily used for values that shouldn't change throughout the program, such as configurations, API keys, or reference values.

Key Characteristics:

- Block-scoped like **`let`**.
- Cannot be reassigned after initialization.
- Must be initialized when declared (i.e., you cannot declare a **`const`** without assigning it a value).

Example:

```
const pi = 3.14159;  // Declare and assign a value
console.log(pi);     // Output: 3.14159

// pi = 3.14;        // TypeError: Assignment to constant variable.
```

Output	3.14159

Mutable Data Example:

If a constant holds an object or an array, its properties or elements can be changed. However, you cannot reassign the entire object or array.

```
const user = { name: "Alice", age: 25 };

// You can modify the properties of the object
user.age = 26;
console.log(user.age);  // Output: 26

// But you cannot reassign the object itself
```

```
// user = { name: "Bob", age: 30 }; // TypeError: Assignment to constant variable.
```

Output	26

5.4. Comparison Between `var`, `let`, and `const`

Feature	`var`	`let`	`const`
Scope	Function or Global	Block	Block
Re-declaration	Allowed	Not allowed in the same scope	Not allowed
Reassignment	Allowed	Allowed	Not allowed
Hoisting	Yes (initialized as undefined)	No	No
Must Initialize at Declaration	No	No	Yes

Best Practices

- Use **`const`** by default when you are sure that the variable's value will not change.
- Use **`let`** when you need to reassign values later in your code.
- Avoid using **`var`** as it can lead to confusing bugs due to its function-scoped nature and hoisting.

6. JavaScript Variable Declaration Rules

6.1 Variable Names

- Must start with a letter, underscore (_), or dollar sign (**$**).
- Cannot start with a number.
- Are case-sensitive.

```
let myVar;     // Valid
let _name;     // Valid
let $price;    // Valid
let var2;      // Valid
let 2price;    // Invalid, cannot start with a number
```

6.2 Reserved Keywords

- Cannot use reserved keywords (e.g., **`let`**, **`var`**, **`if`**, **`else`**) as variable names.

```
let if;     // Invalid, 'if' is a reserved keyword
const var;  // Invalid, 'var' is a reserved keyword
```

6.3 Initialization

- Variables declared with **`let`** or **`var`** can be initialized later.
- Variables declared with **`const`** must be initialized during declaration.

```
let age;         // Declaration without initialization (valid)
const pi = 3.14; // Valid declaration with initialization

const apiKey;    // Invalid, const requires initialization
```

6.4 Unique Names

- **`let`** and **`const`** cannot be redeclared in the same scope.
- **`var`** can be redeclared in the same scope.

```
let score = 10;
let score = 20;  // Invalid, score has already been declared

var count = 5;
var count = 10;  // Valid, var allows re-declaration
```

6. 5 Scope

- **`let`** and **`const`** are block-scoped (limited to the block they're defined in).
- **`var`** is function-scoped or global-scoped.
- Variables declared without **`let`**, **`var`**, or **`const`** become global.

```
function test() {
    if (true) {
        let a = 10;  // Block-scoped
        var b = 20;  // Function-scoped
    }
    console.log(b); // Output: 20
    console.log(a); // ReferenceError: a is not defined (block-scoped)
}

test();

function demo() {
    testVar = 5;  // Implicit global variable (bad practice)
}

demo();
console.log(testVar);  // Output: 5 (accessible globally)
```

7. Primitive Data Types

JavaScript has 7 primitive data types, which represent the simplest forms of values.

7.1 String

- Represents textual data.
- Defined using single quotes (`'`), double quotes (`"`), or backticks (`` ` ``).

```
let name = "John";
let greeting = 'Hello';
let message = `Welcome, ${name}`;
```

7.2 Number

- Represents numeric values (both integers and floating-point numbers).

```
let age = 30;
let price = 9.99;
```

7.3 BigInt

- Represents larger integers.
- Created by appending **n** to the end of an integer.

```
let bigNumber = 9007199254740991n;
```

7.4 Boolean

- Represents a logical value: either **`true`** or **`false`**.

```
let isAvailable = true;
let isLoggedIn = false;
```

7.5 Undefined

- A variable that has been declared but not assigned a value.

```
let x;
console.log(x); // undefined
```

7.6 Null

- Represents the intentional absence of any object value.

```
let y = null;
```

7.7 Symbol

- Represents a unique and immutable value, often used as object property keys.

```
let id = Symbol('id');
```

8. Reference Data Types / Non- Primitive Data Types

In JavaScript, reference data types are non-primitive types, meaning they do not hold the actual data value directly, but instead, they reference a location in memory where the data is stored. These data types are **objects**, **arrays**, and **functions**, which are all treated as **objects** in JavaScript.

Key Characteristics of Reference Data Types:

Stored as references: When a reference data type is assigned to a variable, the variable only holds a reference (or address) to the memory location where the actual data is stored, not the data itself.

Mutable: Reference types are mutable, meaning their values can be changed without changing the reference itself.

Shared references: If two variables point to the same object, changes made to the object via one variable will be reflected in the other.

9. Main Reference Data Types in JavaScript

9.1 Objects

Objects are collections of key-value pairs, where **keys** (also called properties) are strings (or symbols), and **values** can be any data type, including other objects.

```
let person = {
    name: "John",
    age: 30
  };

// Accessing object properties
console.log(person.name);
console.log(person["age"]);
```

Output	John 30

9.2 Arrays

Arrays are a special type of object used to store ordered collections of items **(like a list).** Arrays can contain any type of data (numbers, strings, other objects, etc.).

```
let numbers = [1, 2, 3, 4];
console.log(numbers[0]);

// Arrays can hold different types of values
let mixedArray = [1, "hello", true, {key: "value"}];
console.log(mixedArray[3])
```

Output	1 { key: 'value' }

9.3 Functions

In JavaScript, a **`function`** is a block of code designed to perform a specific task. Functions help organize code into reusable chunks and can be executed whenever they are called or invoked.

Key Characteristics of Functions in JavaScript:

Reusable: Functions can be invoked as many times as needed, with different arguments.

Encapsulation: Functions allow you to encapsulate logic within a defined scope, which makes the code more maintainable and modular.

Parameters: Functions can accept parameters (inputs) and can also return a value after execution.

9.3.1 Syntax of Function

```
function functionName(parameters) {
    // Function body: code to be executed
    return someValue;  // Optional, returns a value to the caller
  }
```

Example:

```
function greet(name) {
    return "Hello, " + name;
  }

let message = greet("John");
console.log(message); // Output: Hello, John
```

Output	Hello, John

- **`greet`** is the **`function`** name.
- **`name`** is the parameter (input).

- The function concatenates **"Hello, "** with the name provided and returns the resulting string.

Functions in JavaScript are also objects. They are instances of the Function type and can be passed as arguments, returned from other functions, or assigned to variables.

```
function greet() {
    console.log("Hello, World!");
  }

let sayHello = greet; // 'sayHello' now references the 'greet' function
sayHello();
```

Output	Hello, World!

10. To-do-List Program | JavaScript

```
// Array to store tasks
var tasks = [];

// Function to add a task
function addTask(task) {
  tasks.push(task);
  console.log("Task added: " + task);
}

// Function to remove a task by its index
function removeTask(index) {
  if (index >= 0 && index < tasks.length) {
    var removedTask = tasks.splice(index, 1);
    console.log("Task removed: " + removedTask);
  } else {
    console.log("Error: Invalid task index.");
  }
}

// Function to display all tasks
function displayTasks() {
  console.log("To-Do List:");
  for (var i = 0; i < tasks.length; i++) {
    console.log((i + 1) + ": " + tasks[i]);
  }
}

// Function to clear all tasks
function clearTasks() {
  tasks = [];
  console.log("All tasks cleared.");
}

// Adding tasks
```

```
addTask("Completing Assignment");
addTask("Study for Quiz");
addTask("Lab Submission");

// Displaying current tasks
displayTasks();

// Removing the second task (index 1)
removeTask(1);

// Displaying tasks after removal
displayTasks();

// Clearing all tasks
clearTasks();

// Displaying tasks after clearing
displayTasks();
```

Output

```
Task added: Completing Assignment
Task added: Study for Quiz
Task added: Lab Submission

To-Do List:
1: Completing Assignment
2: Study for Quiz
3: Lab Submission

Task removed: Study for Quiz

To-Do List:
1: Completing Assignment
2: Lab Submission

All tasks cleared.

To-Do List:
```